Λειτουργικά Συστήματα Υπολογιστών
7ο Εξάμηνο, ΣΗΜΜΥ ΕΜΠ
10/2016

Αναφορά 1ης Άσκησης

Ομάδα E15
Γαβαλάς Νικόλαος 03113121
Καραΐσκου Κωνσταντίνα 03113127

Άσκηση 1.1

Η άσκηση ζητάει να δημιουργηθεί ένα εκτελέσιμο με όνομα zing.
Αρχικά μετακινούμε τα αρχεία zing.h και zing.o που θα χρειαστούμε από το φάκελο /home/oslab/code/zing στον φάκελο εργασίας μας, με τις εντολές (αφού πρώτα "πλοηγηθούμε" στο directory αυτό με την εντολή cd):

```
cp zing.h ~/erg1/ask1       #to "~" einai to $HOME
cp zing.o ~/erg1/ask1
```

Όπου erg1 είναι ένα directory που φτιάξαμε για τα αρχεία αυτής της εργαστηριακής άσκησης.

Στη συνέχεια γράφουμε τη main.c :

```
vim main.c
```

Και στον vi editor γράφουμε:

```
#include<stdio.h>
#include"zing.h"

int main(int argc, char **argv){
        zing();
        return 0;
}
```

Όπως φαίνεται συμπεριλάβαμε το header file που μας δίνεται για να καλέσουμε τη zing()

Ύστερα δημιουργούμε object file κάνοντας compile για τη main.c:

```
gcc -Wall -c main.c
```

Τελικά linkάρουμε τα δύο object files και δημιουργούμε το executable zing με την ακόλουθη εντολή:

```
gcc main.o zing.o -o zing
```

Τρέχοντας το zing εισάγοντας το path στο terminal (χωρίς να έχουμε αλλάξει directory):

```
./zing            #i teleia shmainei auto to directory
```

Παίρνουμε για έξοδο:

```
Hello oslabe15!
```

(Η zing() χρησιμοποιεί τη getlogin(), περισσότερα παρακάτω).

Ερωτήσεις:

1.Ποιο σκοπό εξυπηρετεί η επικεφαλίδα;

Οι επικεφαλίδες είναι αρχεία με προέκταση .h τα οποία περιέχουν πρότυπα και δηλώσεις. Η επικεφαλίδα έχει ως σκοπό τη διεπαφή με άλλα κομμάτια κώδικα (API). Με την εντολή #include μπροστά από την επικεφαλίδα ο μεταγλωττιστής αναγκάζεται να συμπεριλάβει κατά τη φάση της μεταγλώττισης το αρχείο που ορίζει η επικεφαλίδα, το οποίο μπορεί να περιέχει δηλώσεις συναρτήσεων βιβλιοθήκης ή μπορεί να είναι ένα αρχείο που έχουμε φτιάξει εμείς.

2.Ζητείται κατάλληλο Makefile για τη δημιουργία του εκτελέσιμου της άσκησης.

Ανοίγουμε τον vi editor και φτιάχνουμε ένα αρχείο με όνομα makefile, στο οποίο μέσα γράφουμε:

```
updater: main.o zing.o
        gcc -o zing main.o zing.o

main.o: main.c
        gcc -Wall -c main.c
```

Γενικά τα makefiles έχουν την ακόλουθη δομή:

```
target : prerequisites
      Command
```

Οπότε την ακολουθούμε, λέγοντας στο makefile τι εντολές να εκτελέσει και τι αρχεία θα χρειαστεί για να εκτελέσει την καθεμία.

3. Γράψτε το δικό σας zing2.o, το οποίο θα περιέχει zing() που θα εμφανίζει διαφορετικό αλλά παρόμοιο μήνυμα με τη zing() του zing.o. Συμβουλευτείτε το manual page της getlogin(3). Αλλάξτε το Makefile ώστε να παράγονται δύο εκτελέσιμα, ένα με το zing.o, ένα με το zing2.o, επαναχρησιμοποιώντας το κοινό object file main.o.

Γράφουμε τη zing2:

```
 vim zing2.c
```

Και στον vi editor γράφουμε:

```
#include <stdio.h>
#include <unistd.h>

void zing()
{
        printf("How are you %s?\n",getlogin());
}
```

Περιέχει τη διαδικασία zing() που τυπώνει ένα μήνυμα χρησιμοποιώντας την getlogin, η οποία επιστρέφει ένα δείκτη σε συμβολοσειρά που περιέχει το όνομα του συνδεδεμένου χρήστη. Περιλαμβάνουμε και το header file που περιέχει τη συνάρτηση getlogin. Η διαδικασία καλείται πάλι από το object file main.o.

΄Υστερα δημιουργούμε object file κάνοντας compile για τη zing2.c:

```
gcc -Wall -c zing2.c
```

Στη συνέχεια κάνουμε link τα δύο object files (main.o, zing2.o) και δημιουργούμε εκτελέσιμο zing2 με τον ακόλουθο τρόπο:

```
gcc main.o zing2.o -o zing2
```

Τρέχοντας το zing2 παίρνουμε την ακόλουθη έξοδο:

```
How are you oslabe15?
```

Τέλος, φτιάχνουμε ένα αρχείο με το όνομα makefile στο οποίο γράφουμε:

```
updater1: main.o zing.o zing2.o
        gcc -o zing main.o zing.o
        gcc main.o zing2.o -o zing2

zing2.o: zing2.c
        gcc -Wall -c zing2.c

main.o: main.c
        gcc -Wall -c main.c
```

Έτσι παράγονται δύο εκτελέσιμα αρχεία, τα zing και zing2.

4. Έστω ότι έχετε γράψει το πρόγραμμά σας σε ένα αρχείο που περιέχει 500 συναρτήσεις. Αυτή τη στιγμή κάνετε αλλαγές μόνο σε μία συνάρτηση. Ο κύκλος εργασίας είναι: αλλαγές στον κώδικα, μεταγλώττιση, εκτέλεση, αλλαγές στον κώδικα, κ.ο.κ. Ο χρόνος μεταγλώττισης είναι μεγάλος, γεγονός που σας καθυστερεί. Πώς μπορεί να αντιμετωπισθεί το πρόβλημα αυτό;

Χρησιμοποιώντας απλά makefile. Τα makefiles εκτελούν μονο τις εντολές των οποίων τα prerequisites έχουν αλλάξει, με αποτέλεσμα να σώζεται πολύς χρόνος. Συνεπώς, αν κάναμε αλλαγές σε μία μόνο συνάρτηση, το makefile θα εκτελούσε τις εντολές που αφορούν μόνο αυτή, ενώ τις άλλες θα τις κρατούσε ίδιες από προηγούμενο make.

5. Ο συνεργάτης σας και εσείς δουλεύατε στο πρόγραμμα foo.c όλη την προηγούμενη εβδομάδα. Καθώς κάνατε ένα διάλειμμα και ο συνεργάτης σας δούλευε στον κώδικα, ακούτε μια απελπισμένη κραυγή. Ρωτάτε τι συνέβει και ο συνεργάτης σας λέει ότι το αρχείο foo.c χάθηκε! Κοιτάτε το history του φλοιού και η τελευταία εντολή ήταν η: gcc -Wall -o foo.c foo.c Τι συνέβη;

Η εντολή που χρησιμοποιήθηκε δημιούργησε executable, με όνομα “foo.c”, και διέγραψε το υπάρχον foo.c, γιατί πρακτικά έγραψε πάνω του αντικαθιστώντας το με το εκτελέσιμο του foo.c. Καλό είναι γι αυτό να χρησιμοποιούμε makefiles και να κρατάμε backups.

Άσκηση 1.2

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

size_t idx=0; //το index πρεπει να ειναι global (δείχνει σε ποιο σημείο του αρχείου είμαστε)

void doWrite(int fd, const char *buff, int len) //γραφει στον fd
{
        ssize_t wcnt;
        do{
                wcnt= write(fd, buff+idx, len-idx);
                if (wcnt == -1) { /*error*/
                        perror ("write");
                        exit(1);
                }
                idx += wcnt;
        }while(idx<len);
}

void write_file (int fd, const char *infile)
{
        int fd1;
        fd1= open(infile, O_RDONLY);  //ανοιγμα αρχειου για διαβασμα
        if (fd1==-1) {
                perror ("open");
                exit (1);
        }
        char buff[1024];
```

```
        ssize_t rcnt;
        size_t len;
        idx=0;
        for(;;) {
                rcnt= read(fd1, buff, sizeof(buff)-1); //διάβασμα αρχείου
                if (rcnt==0) /*end of file*/
                        break;
                if (rcnt==-1) { /*error*/
                        perror ("read");
                        exit(1);
                }
                buff[rcnt]= '\0';
                len= strlen (buff);       //το len περιέχει το μέγεθος του
κειμένου
                doWrite (fd, buff, len);  //κλήση της doWrite

        }
        close(fd1);
}

int main(int argc, char **argv)  //τρόπος λειτουργίας: A (append)-> C, B
(append) -> C

{
        if (argc<3){     //αν τα arguments (infiles) ειναι λιγοτερα απο
2, τύπωσε οδηγιες
                printf("Usage: ./fconc infile1 infile2 [ outfile (default: fconc.out )
]\n");
                return -1;
        }
        char *outfile;
        if (argc==3){   //αν δεν παρατίθεται outfile χρηση fconc.out ως
default
                outfile="fconc.out";
        }
        if (argc==4){   //αλλιώς χρησιμοποιείται το 3ο όρισμα για
outfile
                outfile=argv[3];
        }
        int fd;
        int oflags = O_CREAT | O_WRONLY | O_APPEND; //η σημαία append είναι
πολύ
        int mode = S_IRUSR | S_IWUSR;           //σημαντική.Μας επιτρεπει
να γραφουμε σε
        fd = open(outfile, oflags, mode);     //ενα αρχείο συμπληρωνοντας
στο τελος του.
        if (fd == -1){
                perror("open");
                exit(1);
        }
```

```
        write_file (fd, argv[1]); //κλησητης write_file
        write_file (fd, argv[2]);
        close(fd);
        return 0;
}
```

Η strace αν εκτελέσουμε **strace ./fconc a b c** δίνει:

```
execve("./fconc", ["./fconc", "a", "b", "c"], [/* 19 vars */]) = 0
brk(0)                                  = 0x94fa000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7790000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=95992, ...}) = 0
mmap2(NULL, 95992, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7778000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\200\230\1\0004\0\0\0"...,
512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1516996, ...}) = 0
mmap2(NULL, 1522204, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0xb7604000
mmap2(0xb7772000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x16e000) = 0xb7772000
mmap2(0xb7775000, 10780, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0xb7775000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7603000
set_thread_area({entry_number:-1, base_addr:0xb7603940, limit:1048575, seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
(entry_number:6)
mprotect(0xb7772000, 8192, PROT_READ)   = 0
mprotect(0xb77b4000, 4096, PROT_READ)   = 0
munmap(0xb7778000, 95992)               = 0
open("c", O_WRONLY|O_CREAT|O_APPEND, 0600) = 3
open("a", O_RDONLY)                     = 4
read(4, "This is file A\n", 1023)       = 15
write(3, "This is file A\n", 15)        = 15
read(4, "", 1023)                       = 0
close(4)                                = 0
```

```
open("b", O_RDONLY)                     = 4
read(4, "This is file B\n", 1023)       = 15
write(3, "This is file B\n", 15)        = 15
read(4, "", 1023)                       = 0
close(4)                                = 0
close(3)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Με bold φαίνονται παραπάνω τα σημεία που φανερώνουν ξεκάθαρα τον τρόπο λειτουργίας του προγράμματος. Πρώτα ανοίγουμε τα το outfile για γραψιιμο, ύστερα διαβάζουμε το πρώτο, το γράφουμε στο outfile, και μετά διαβάζουμε το 2ο και το κάνουμε και αυτό write με append στο 3ο.

Σε ό,τι αφορά την εκτέλεση της fconc σε corner cases:

Αν τρέξουμε την fconc με κάποιο από τα infiles να είναι και το outfile, π.χ. `./fconc a b b` θα περιμέναμε να γραφτεί το a στο b και ύστερα το b στον εαυτό του, οπότε αν το αρχικό περιεχόμενο του a ήταν “This is A” και του b ήταν “This is B”, τότε αφού εκτελούσαμε την εντολή θα περιμέναμε να έχει τελικά το b εν τέλει:

This is B
This is A
This is B
This is A

γιατί πρώτα γράφτηκε το a στο b, (που είχε ήδη μέσα του “This is B”) και μετά το b έγινε append στον εαυτό του. Εκτελώντας τελικά την εντολή διαπιστώνουμε ότι έτσι συμβαίνει πράγματι.